A Simple Convention for the Specification of Linear Algebra Function Prototypes in C++

ROSCOE A. BARTLETT
Sandia National Laboratories, Albuquerque NM 87185 USA

This short note describes a simple convention for the specification of C++ function prototypes for linear algebra operations with vectors and matrices (or general linear operators). This convention leads to function prototypes that are derived directly from the mathematical expressions themselves (and are therefore easy to remember), allow for highly optimized implementations (through inlining in C++), and do not rely on any sophisticated C++ techniques so that even novice C++ programmers can understand and step through the code in a debugger.

Categories and Subject Descriptors: ... [...]: ...

General Terms: Algorithms, Design, Performance, Standardization

Additional Key Words and Phrases: Object-Orientation, Vectors, Interfaces, C++, ...

1. INTRODUCTION

Linear algebra computations such as matrix-vector multiplication and the solution of linear systems serve as the building blocks for numerical algorithms and consume the majority of the runtime of numerical codes. These linear algebra abstractions transcend details such as matrix storage formats (of which there are many) and linear system solver codes (sparse or dense, direct or iterative). Primary linear algebra abstractions include vectors and matrices and the operations that can be performed with them. C++ abstractions for vectors and matrices abound.

Once convenient vector and matrix classes `Vec` and `Mat` are created, for instance, there is a need to implement BLAS-like linear algebra operations. Given that C++ has operator overloading, it would seem reasonable to implement these operations using a MATLAB© like notation. For example, the matrix-vector multiplication $y = y + A^T x$ might be represented in C++ with the statement `y = y + trans(A)*x` (the character `'` can not be used for transpose since it is not a C++ operator). MATLAB is seen by many in the numerical computational community to be the ideal for the representation of linear algebra operations using only ASCII characters [Demmel, J. 1997]. The advantages of such an interface are obvious; It is almost the same as standard mathematical notation, which makes it very easy to match the implementation with the operation for the application programmer, and makes

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

the code much easier to understand. The primary disadvantage for this approach in C++ is that the straightforward implementation requires a lot of overhead because operators are implemented in a pair-wise fashion involving temporaries. For example, for the operation `y = y + trans(A)*x`, a temporary matrix (n^2 overhead) and two temporary vectors ($2n$ overhead) would be created by the compiler. Specifically, the compiler would perform the following operations: `Mat t1 = trans(A); Vec t2 = t1*x; Vec t3 = y + t2; y = t3;`. Attempts have been made to come up with a strategy in C++ to implement operations like `y = y + trans(A)*x` in a way where little overhead is required beyond a direct BLAS call [Parker, B. 1997]. It is relatively easy to implement these operator functions with only a little constant-time overhead for a small set of linear algebra operations [Stroustrup, B. 1997, pages 675-677]. However, for more elaborate expressions, a compile-time expression parsing method is needed. Some have advocated preprocessing tools, while others have looked at using C++'s template mechanisms [Veldhuizen, T. and E. Gannon 1998], [Parker, B. 1997]. In any case, these methods are complex and not trivial to implement. Methods based on runtime parsing are also possible but add more of a runtime penalty. Aliasing is also another big problem. For example, suppose we allow users to write expressions like

$$y = x + v + \alpha M^T + \beta y.$$

An efficient parser that tries to minimize temporaries will have to scan the entire expression and realize that $y = \beta y$ must be performed first and then no temporaries are needed. A naive parser may perform $y = x$ first and then result in an incorrect evaluation. The problem is that the more efficient the parser the more complicated it is and the harder it will be for inexperienced users to debug through this code.

Another problem is that operator-overloading implementations in C++ can never generate error messages of the same quality as MATLAB. Consider an expression of the form `r = A*x + B*y + C*z` where the matrix `B` and the vector `y` are mismatched. Since MATLAB is an interpretive language, the MATLAB interpreter can give a very good error message that gives the file name, the line number and even the statement that caused the error. Generating this type of error using operator overloading in C++ is generally not possible. One would have at the very least to open a debugger, set a breakpoint, and then step back up the call stack in order to get the same information. Therefore, operator overloading using the current C++ standard will never achieve the same level of usability as operator syntax in MATLAB.

Without using operator overloading to allow application code to use syntax like `y = y + trans(A)*x`, how can linear algebra operations be implemented efficiently? The simple answer is to use regular functions (member or non-member) inlined to call BLAS-like implementations. For example, for the operation $y = y + A^T x$, one might provide a function like `add_to_multiply_transpose(A,x,&y);`. It is trivial to implement such a function to call the BLAS, for instance, with no overhead if a good inlining C++ compiler is used. The problem with using functions is that it is difficult to come up with good names that users can remember. For example, the above operation has been called `Blas_Mat_Vec_Mult(...)` in LAPACK++ [Pozo, R. 1996], `vm_multadd(...)` in Meschach++ [Roberts, S., et. al. 1996], and `mult(...)` in MTL [Lumsdanie, A. and J. Siek 1998]. Even knowing the names of

Function Call

```
y  +=  alpha * A' * x    y     α       A^T     x
↓  ↓     ↓   ↓ ↓ ↓ ↓     ↓     ↓       ↓      ↓
V  p_    S   t M t V ( &y, alpha, A, TRANS, x ) ⟹ Vp_StMtV(&y,alpha,A,TRANS,x)
```

Function Prototype

```
void Vp_StMtV( Vec* y, Scalar alpha, const Mat& A, Transp A_trans, const Vec& x);
```

Fig. 1. Example of the linear algebra naming convention for $y{+}{=}\,\alpha A^T x$

these functions is not enough. You must also know the order the arguments go in and how are they passed.

ToDo: Mention recent ACM TOMS article

2. A CONVENTION FOR SPECIFYING FUNCTION PROTOTYPES

Here we consider a convention for constructing C++ function prototypes. Function prototypes are constructed according to this convention where the name of the function and the order of the arguments is easily composed from the mathematical expression itself. To illustrate the convention, consider the operation $y = y+\alpha A^T x$. First, rewrite the operation in the form $y+= \alpha A^T x$ (this is well understood by C, C++ and Perl programmers). Next, translate into MATLAB-like notation as `y += alpha*A'*x` (except MATLAB does not have the operator `+=`). Finally, for `Vec` objects `y` and `x` and a `Mat` object `A`, the function call and its prototype are shown in Figure 1. In this function prototype, type `Transp` is a simple C++ `enum` with the values `TRANS` or `NOTRANS`, and the type `Scalar` can be a simple typedef to `double` or might be a template argument.

A summary of this convention is shown in Figure 2. Given this convention, it is easy to go back and forth between the mathematical notation and the function prototype. For example, consider the function call and its mathematical expression

```
Mp_StMtM( &C, alpha, A, NOTRANS, B, TRANS )
```

$\Longrightarrow$

$C+= \alpha A B^T$.

One difficulty with this convention is dealing with Level-2 and Level-3 BLAS that have expressions such as

$$C = \alpha\,\mathrm{op}(A)\,\mathrm{op}(B) + \underbrace{\beta}_{?} C \qquad \text{(xGEMM)}.$$

Given $\beta \neq 1$ we can not simply rewrite the above BLAS operation using $+=$. To deal with this problem, β is moved to the end of the argument list and has a default value of 1.0 as

Operation	Character (Lower Case)
=(assignment,equals)	_(underscore)
+=(plus equals)	p_
-=(minus equals)	m_
*=(times equals)	t_
+(addition,plus)	p
-(subtraction,minus)	m
*(multiplication,times)	t

Operand Type	Character (Upper Case)	Argument(s)
Scalar	S	Scalar
Vector	V	(rhs) const Vec&
		(lhs) Vec*
Matrix	M	(rhs) const Mat&, Transp
		(lhs) Mat*

Fig. 2. Naming convention for linear algebra functions in C++

```
Mp_StMtM( &C, alpha, A, A_trans, B, B_trans ,      beta      ).
                                                 default to 1.0
```

Only exact equivalents to the Level-2 and Level-3 BLAS need be explicitly implemented (i.e. `Vp_StMtV(...)` and `Mp_StMtM(...)`). Functions for simpler expressions can be generated automatically using template functions. As an example, consider the linear algebra operation and its function call

$$y = Ax \qquad (\text{xGEMV} \rightarrow y = \alpha\,\text{op}(A)x + \beta y)$$
$$\Longrightarrow$$
```
V_MtV( &y, A, NOTRANS, x ).
```

In the above example, the template function `V_MtV(...)` can be inlined to call `Vp_StMtV(...)` which in turn can be inlined to call the BLAS function `DGEMV(...)`, for instance. The use of these automatically generated functions makes the application code more readable and also allows for specialization of these simpler operations later if desired. The implementation of the above template function `V_MtV(...)` is trivial and is

```
template<class M, class V>
inline void V_MtV( V* y, const M& A, Transp A_trans, const V& x )
{
  Vp_StMtV( y, 1.0, A, A_trans, x, 0.0 );
}
```

Similar templated functions can also be generated for the partial simplifications `Vp_MtV(...)` and `V_StMtV(...)`.

Longer expressions such as $y = \alpha A^T x + Bz$ are easily handled using multiple function calls such as

$y = \alpha A^T x + Bz$

$\Longrightarrow$

```
V_StMtV( &y, alpha, A, TRANS, x );
Vp_MtV( &y, B, NOTRANS, z );
```

As stated above, only the base BLAS operations `Vp_StMtV(...)` (e.g. `xGEMV(...)`) and `Mp_StMtM(...)` (e.g. `xGEMM(...)`) must be implemented for the specific vector and matrix types `Vec` and `Mat`. For example, if these are simple encapsulations of BLAS compatible serial vectors and matrices (e.g. TNT style) then the call the the BLAS functions can be written as template functions for all serial dense vector and matrix (column oriented) classes. For example, an inlined call to `DGEMV(...)` can be performed as

```
template<class M, class V>
inline void Vp_StMtV( V* y, Scalar alpha, const M& A, Transp A_trans
  , const V& x, Scalar beta = 1.0 )
{
  DGEMV( A_trans == NOTRANS ? 'N' : 'T', rows(A), cols(A), alpha
    ,&A(1,1), &A(1,2) - &A(1,1), &x(1), &x(2) - &x(1), beta
    ,&(*y)(1), &(*y)(2) - &(*y)(1) );
}
```

Of course the above function would also have to handle the cases where `rows(A)` and/or `cols(A)` was 1 but the basic idea should be clear. By calling `rows(...)` and `cols(...)` as nonmember functions, they can be overloaded to call the appropriate member functions on the matrix object since there is no standard names for these.

When `Vec` and `Mat` are polymorphic types, one can use a common trick to implement `Vp_StMtV(...)` and `Mp_StMtM(...)` using member functions. For example, `Vp_StMtV(...)` can be implemented as

```
class Vec { ... };

class Mat {
public:
  virtual void apply( Vec* y, Scalar alpha, Transp A_trans
    , const Vec& x, Scalar beta ) const = 0;
  ...
};

inline void Vp_StMtV( Vec* y, Scalar alpha, const Mat& A, Transp A_trans
  , const Vec& x, Scalar beta = 1.0 )
{
  A.apply(y,alpha,A_trans,x,beta);
}
```

Using these inlined non-member functions there is no extra overhead beyond the unavoidable virtual function calls. In this way there is consistent calling of linear algebra operations irregardless whether the vector and matrix objects are concrete or abstract.

This convention can, of course, also be used for level-1 vector-vector operations like `Vt_S(&y,alpha)` for $y = \alpha y$, `V_V(&y,x)` for $y = x$, `Vp_StV(...)` for $y = y + \alpha x$, `V_VpV(&z,x,y)` for $z = x + y$, and `V_VmV(&z,x,y)` for $z = x - y$. One can write as many of these convenience wrapper functions as desired.

3. CONCLUSIONS

In summary, this convention makes it easy to write out correct calls to linear algebra operations without having to resort to complex operator-overloading techniques. After all, the main appeal for operator overloading is to make it easy for users to remember how the call linear algebra operations and to make written code easier to read and interpret. The convention described in this note meets both of these goals and also results in code that is easy for novice C++ developers to understand and debug through. Debugging code can easily take longer than writing it in the first place. When concrete abstractions of dense linear algebra types are used, it was shown that these functions do not have to impose any overhead beyond direct BLAS calls if inlining is used. When polymorphic vector and matrix types are used, inlining to call the virtual functions also results in no extra overhead. This convention has been used in the development of MOOCHO [**?**] and in Thyra [**?**].

REFERENCES

Demmel, J. 1997. *Applied Numerical Linear Algebra*. SIAM.

Lumsdanie, A. and J. Siek. 1998. The Matrix Template Library. http://www.lsc.nd.edu/research/mtl/.

Parker, B. 1997. Template Composite Operators. http://www.gil.com.au/bparker.

Pozo, R. 1996. *LAPACK++ v 1.1: High Performance Linear Algebra User's Guide*. NIST.

Roberts, S., et. al. 1996. Meschach++: Matrix Computations in C++. http://www.netlib.org/c/meschach/.

Stroustrup, B. 1997. *The C++ Programming Language, 3rd edition*. Addison-Wesley, New York.

Veldhuizen, T. and E. Gannon. 1998. Active Libraries: Rethinking the Roles of Compilers and Libraries. http://oonumerics.org/blitz/papers/.

Received: ???; revised: ???; accepted: ???